Gustavo Cornejo
CS165A

Architecture

My solution was written in python 2.7.11 as it's the default version of python on CSIL. The architecture of my solution is comprised of four classes. The first is a parent class called Scrapers, this class was made with the purpose of providing common functions, variables that would be needed when reading from the files. This can either be when first creating and training the naïve bayes classifier or when you're reading from the testing.txt file, in either case we would need this class. In this class we have a few class variables that provide a way to open the file, stop words, symbols we need to parse out, and counters. In addition, it includes two functions is_positive_review, to help categorize documents by checking if they end with a 1 or 0, and sentence_pre_process which helps us get rid of any special characters in the file we are reading.

The next class is BuildSet which inherits from its parent class, Scrapers. Here the purpose was to create a class that would handle both the creation and the training of the naïve bayes classifier. It's main function, build_set , is the main component that reads and parses through the file while add_to_dict helps it append values to a respective dictionary. Within this class we also have two variables posWords and negWords, these represent the respective words and their frequencies that appear in either positive or negative reviews.

The other class that inherits from Scrapers is the Classifier class. This class handles processing the testing.txt file in its main function and creating Document objects for every line in the file. The class also handles several class variables from the BuildSet class that are brought in during instantiation that are later passed into the Document objects for calculations. Such as the frequency counts for words that were built during the learning stage and the total number of docs in the classifier of either good or bad labels.

The final class is the Document class, this is where the document string is parsed and the calculation for each document is made to determine if it belongs in the positive or negative review class. The function create_word_list parses the document string and creates a word list. Then calculate_score and set_calculated_score do the math behind determining the label for the document. This class does require information from the model, for the calculation, during instantiation. This is passed from the BuildSet class to the Classifier class and ultimately to the Document object when it's created.

Preprocessing

In the preprocessing stage of my solution, I focused on cleaning up each line of text that was read in from the file. Within the build_set function, in the BuildSet class, I run each line of text through a function called sentence_pre_process. In this function I parse out any special characters like "?","!","<br />", etc. In addition, before adding any word to my model, I get rid of any non-essential words that relate no actual information about the nature of the movie review. These words include “movie”, “film”, “this”, “of”, and etc. Through these methods I was able to end up with a clean version of any document that was read in the training.txt file. Afterwards I represented each term in the documents in a python dictionary as a key, value pair where the string representation of the term would be the key and its frequency would serve as the value. I set up two dictionaries, posWords & negWords, to register terms that appear in either positive or negative reviews of the training.txt file. Here each term would represent a new feature of either the positive or negative review class.

Model Building

In order to train the Naïve Bayes Classifier, I first ran each line of the training.txt file through the preprocessing discussed in the previews section. Following this step, I went through each cleaned up version of the documents and identified whether the document string ended with a 1 or a 0. This of course was to find out the intended category for this particular document and to keep count of the ration between positive and negative reviews in our training data set. Afterwards, for each document, I counted the frequencies of each word and registered it in the positive review python dictionary, posWords, if it ended with a 1, or the negative review python dictionary, negWords, if it ended with a 0. In the case that the word belonged to the group of stop words I had defined, the word was not included in the dictionaries. After building the model, the dictionaries represented a list of words that appeared in positive and negative reviews, accompanied by its frequency in each respective category.

Results

After building my models, I achieved an accuracy of 0.9794 on the training.txt file with a runtime of 4 seconds. While on the testing.txt file I reached an accuracy of 0.842 with a runtime of 2 seconds. In my solution the most important features ultimately were decided by their frequency. For the positive class the most important features included the words “not,it’s,out,about,very,or,good,great,well,more”. For the negative class the most important features were, “not, or, just, if ,about, no,bad,only,more,even,out”.

Challenges

The main challenge that I had faced throughout this assignment was achieving an acceptable accuracy. In prior versions of my solution I was calculating probabilities without applying smoothing correctly which set my accuracy around the .60 mark. My incorrect interpretation of smoothing was only adding the +1 shown below to words that did exist in my vocabulary created from the training data. This of course meant that I was not actually avoiding overfitting. The other error I was making was that instead of adding the vocabulary size in the denominator, I was adding the number of positive/negative review documents which was incorrect.

$$\hat{P}(x_i \mid c_j) = \frac{N(X_i = x_i, C = c_j) + 1}{N(C = c_j) + k}$$

# of values of X_i

The other challenge I was facing was floating point underflow in my calculation of probabilities. This was easily solved after switching my implementation to the following specifications:

$$c_{NB} = \underset{c_j \in C}{\operatorname{argmax}} \log P(c_j) + \sum_{i \in positions} \log P(x_i \mid c_j)$$

After the changes were made I was at a more respectable .84 accuracy rating. Other challenges that I faced revolved around providing a simple and efficient design to my solution that would allow for a low runtime and great code readability. I simply took time to draw out the different components and developed classes around each component and its main task. Following this it was only a matter of picking the correct data structure, python dictionary, to represent the frequency counters.

Weaknesses

One of the main weaknesses of my solution is the fact that I don't address stop words in an efficient and fully effective manner. This is evidenced by the fact that the top words in both my positive and negative classes contain words like "about", "if,", and "more" that don't really provide any substance to the document. My current solution is a simple array that includes only some stop words and is used to avoid including these words in my model. However, perhaps a more efficient way of doing this would be to not only add more stop words to my array, uselessWords, but also to get rid of any words that have a high count in both the positive and negative classes as they are not as indicative as words with a distinct difference in positive/negative ratio.

Another avenue that I could explore to improve my solution is to include word stemming. By grouping together words by their roots, I make prefixes and suffixes irrelevant. This technique would create more realistic representations of words in my model, which in turn provide a better probabilistic outcome in determining a documents association to the positive and negative review classes.